Scheduling on hierarchical clusters using malleable tasks

Pierre-François Dutot, Denis Trystram

Scheduling on hierarchical clusters using Malleable Tasks

Pierre-François Dutot and Denis Trystram
ZIRST, 51 avenue Jean Kuntzmann
38330 Montbonnot Saint Martin, France
contact author: Denis.Trystram@imag.fr

January 26, 2001

Submitted as a regular presentation to SPAA'01

Abstract

The model of *malleable task* (MT) was introduced some years ago and has been proved to be an efficient way for implementing parallel applications. It considers a target application at a larger level of granularity than in other models (corresponding typically to numerical routines) where the tasks can themselves be executed in parallel.
Clusters of SMP (symmetric Multi-Processors) are a cost effective alternative to parallel supercomputers. Such hierarchical clusters are parallel systems made from m SMP composed each by k identical processors. They are more and more popular, however, designing efficient software that take full advantage of such systems remains difficult. This work describes a $2 - \frac{2}{k}$ approximation algorithm for scheduling a set of independent malleable tasks for the minimization of the parallel execution time, where k is a power of 2 ($k \geq 2$). For $k = 2$, a special treatment leads to the bound of $3/2$ which is the best known for non hierarchical tasks.

1 Introduction

Until recently, the standard communication model for scheduling the tasks of a parallel program has been the *delay* model introduced by Rayward-Smith for UET-UCT task graphs (unit execution times and unit communication times) [16]. In this model, the communications between tasks allocated to different processors are considered explicitly by the transmission time of a message between them. The scheduling problem with communication delays is known to be NP-hard in the strong sense [16]. However, efficient heuristics have been developed for coarse grain computations, i.e. when communication times are smaller than computation times. Unfortunately, this hypothesis is not often valid in practice, especially for large parallel clusters or grid computing. A good alternative to deal with communications is to consider the *malleable tasks* model where the communication times are considered implicitly by a function representing the penalty on the workload due to the management of the parallelism. As detailed later, this function will be considered as *monotonic*, i.e. it is increasing with the number of processors, whereas the execution time decreases, at least until a certain threshold. More formally, a malleable task is a computational unit which may be executed on several processors with a running time that depends on the number of processors allotted to it. Some authors used recently this model for parallelizing actual applications [2, 6].

Cluster computing addresses a new trend of high-performance computing in which a large set of PCs are linked via a fast interconnection network in order to obtain a huge power at a reasonable price. The concept of cluster computing was developed within the *Beowulf* project which was originated from NASA [18]. The number of cluster installations around the world has grown significantly in the last few years. Such systems are very popular since they provide large memory, scalability and fault tolerance. Many of them have a hierarchical structure where the nodes are themselves SMP (symmetric multi-processors) with 2, 4 or 8 processors [5].

In this paper, we are interested in scheduling a set of n independent malleable tasks on a cluster hierarchical system of m SMP composed each by k identical processors. The objective is to minimize the *makespan*, i.e. the minimum completion time. According to most existing clusters, we consider that k is a power of 2. The main contribution of this work is to propose a new approximation algorithm for scheduling independent malleable tasks with a performance guarantee of $2 - \frac{2}{k}$ for $k > 2$. Therefore, for $k = 4$, we obtain the same bound ($\frac{3}{2}$) as for non hierarchical systems [15]. In the particular case of $k = 2$ we will show also that this bound can be reached.

The problem of scheduling a set of independent parallel jobs have been considerably studied in the last few years [7]. The main distinction between these works is to consider static schedules or dynamic ones. We focus here on static schedules, corresponding for instance to batch scheduling, where a set of tasks together with an estimation of their execution times are known before their execution.

The problem of scheduling malleable tasks on usual parallel distributed systems is known to be NP-hard, even for independent tasks, as a generalization of the classical multiprocessors scheduling problem [8]. Jansen & Porkolab [11] proposed a fully polynomial approximation scheme based on a Integer Linear Programming formulation for scheduling independent malleable tasks. The complexity of the scheme, although linear in the number of tasks, is high independently of the accuracy of the approximation due to an exponential factor in the number of processors. Thus, even if the result has some theoretical interest, the heuristic cannot really be used in practice. We are interested in efficient, low complexity, heuristics with good performance guarantee. Most existing works are based on a two-phases approach introduced by Turek, Wolf and Yu [19]. The basic idea is to select in a first step an allotment (the number of processors allotted to each task), and then solve the resulting non-malleable scheduling problem, which is a classical multiprocessor scheduling problem. As far as the makespan criterion is concerned, this problem is identical to a 2-dimensional strip-packing problem [1, 4]. It is clear that applying an approximation of guarantee λ for the non-malleable problem on the allotment of an optimal solution provides the same guarantee λ for the malleable problem. Ludwig [12] improved the complexity of the allotment selection of the Turek's

algorithm in the special case of monotonic tasks. Based on this result and on the 2-dimensional strip-packing algorithm of guarantee 2 proposed by Steinberg [17], he presented a 2-approximation algorithm for the malleable scheduling problem. Mounié, Rapine and Trystram improved this result by concentrating more on the first phase (the allotment problem). More precisely, they proposed to select an allotment such that it is no more needed to solve a general strip-packing instance, but a simpler one where better performance guarantee can be ensured. They obtained a $\sqrt{3}$-approximation algorithm [14] and more recently a $\frac{3}{2}$-approximation algorithm [15].

Some works studied the scheduling problem of general graphs with precedence constraints for hierarchical systems [3]. Giroudeau established an non-approximability result for this problem in the particular case of identical tasks on a hierarchical system with an unbounded number of processors assuming unit communication times between SMPs [9]: no algorithm can be found with a better bound than $\frac{5}{4}$.

The paper is organized as follows: the malleable task model and some basic assumptions are presented in section 2 and some definitions are recalled. Section 3 aims to describe the principle of the algorithm, based on two phases of allotment into two partial shelves and scheduling inside the shelves. Section 4 presents the allotment of the tasks, and section 5 some transformations which may be applied to this allotment. Then, the feasibility is analyzed in section 6. The scheduling is presented in section 7, and the algorithm and its complexity are in section 8. Section 9 details the special case $k = 2$ of clusters of biprocessors. Finally, we conclude by discussing some promising perspectives and open problems.

2 Malleable task Model

The model of computation that we consider in this work is such that a processor can compute only one task at a time. We also assume that the number of processors allocated to a task is constant during its whole execution. We are looking for non-preemptive schedules, however, the performance guaranties are established in respect to an optimal solution, which maybe preemptive.

We consider an instance composed of n malleable tasks $\{T_1, \ldots, T_n\}$ to be scheduled on a cluster of m SMP composed each of k identical processors, where k is a power of 2 ($k = 2^q \geq 2$). The execution time of the malleable task T_i when allotted to p processors will be denoted by $t_{i,p}$. Its *computational area* (or *work*) is defined as usually as the time space product $W(i,p) = pt_{i,p}$. The total work $W = \sum W(i,1)$ divided by mk is a straightforward lower bound of the optimal makespan.

We use in this paper the dual approximation techniques introduced by Hochbaum & Shmoys [10]. Given a real number d, a dual λ-approximation either delivers a schedule of length at most λd, or concludes (correctly) that no schedule of length lower than d exists.

By using a dichotomic search, a λ-dual approximation of complexity $\mathcal{O}(f(n))$ can be converted into a $\lambda(1+2^{-s})$-approximation of complexity $\mathcal{O}(sf(n))$ for any integer s.

Therefore let d be a real number. Since we will "guess" the value of the optimal makespan, we introduce the *canonical allotment* defined by the minimal number of processors (denoted p_i^d) for executing task T_i in time less than d.

Using more processors will cost some penalty for managing the communications and synchronizations. According to the usual behavior of the execution of parallel programs, we assume that the tasks are *monotonic*. This means that allocating more processors to a task will decrease its execution time and increase its computational area. We now state two fundamental properties which are two direct consequences of the monotonic hypothesis.

Property 1 *If p_i^d exists, then $t_{i,p_i^d} > \frac{p_i^d - 1}{p_i^d} d$.*

This property is straightforward by simply writing the decreasing of the computational area. Indeed we have $p_i^d t_{i,p_i^d} \geq (p_i^d - 1) t_{i,p_i^d - 1}$. Simply notice that by definition of p_i^d we have $t_{i,p_i^d - 1} > d$. □

Property 2 *Assume that there exists a schedule of length lower than d. Then for any allotment q_i such that $q_i \leq p_i^d$ for any i, we have $\sum_{i=1}^{n} q_i t_{i,q_i} \leq mkd$*

If such a schedule exists, clearly any task T_i is allocated to at least p_i^d processors. Due to the monotony

of the computational area, the allotment using the q_i's has a total area lower than the one of the p_i's. However this last area is bounded by mkd, since the makespan of the schedule does not exceed d. □

In the following we will normalize the given problem by dividing all the execution times by d, the guessed optimal makespan. Therefore, without loss of generality, the optimal makespan will always be assumed to be 1 in the new problem.

There exists a difficulty inherent to hierarchical systems due to the fact that communications inside the same SMP are faster than between processors belonging to different SMP. In this case, the number of processors allotted to a task does not give all the informations needed to determine the execution time of a task: a task will be scheduled faster using processors inside the same SMP than using processors of different SMP. In order to avoid this problem, we introduce below a dominant rule:

Definition 1 *(Best placement rule)*
For a given number of processors, we say that a task is in its best placement if the penalty with this number of processors is the lowest as possible.

This definition is not very useful in the sense where many placements may verify the *best placement* condition, and from the definition we cannot decide where it is the best to schedule the task. However, we can make the assumption that a task which runs on less than k processors will be in the *best placement* state if all the processors allotted to the task are into the same SMP.

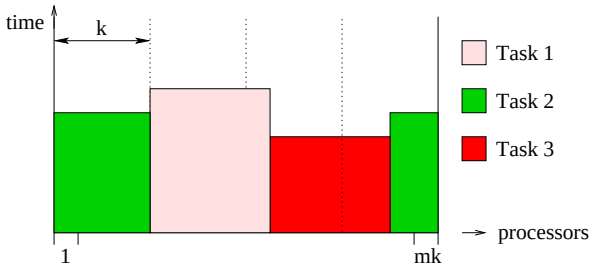

Figure 1: Tasks 1 and 2 are in their *best placement*, whereas task 3 is not ($m = 4$).

Hypothesis 1 *(Minimal penalty)*
We assume in the rest of the paper that a task T_i allotted on $a_i k + b_i$ processors (with $a_i \in [0; m]$ and $b_i \in [0; k-1]$) is in its best placement if exactly a_i SMPs are dedicated to it during its execution and the remaining b_i processors are within the same SMP.

Remark that we do not ask the processors to be contiguous. For instance, figure 1 represents two tasks verifying the *minimal penalty* hypothesis. The third one does not.

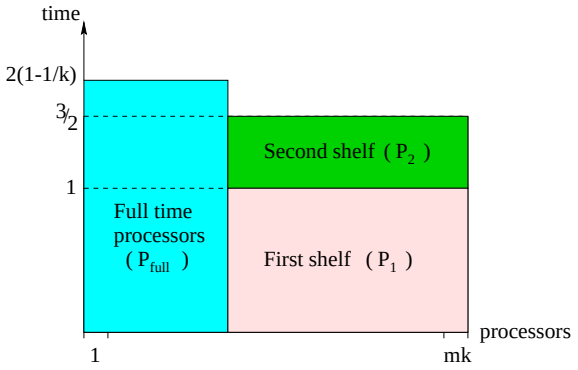

Figure 2: Shape of the schedule.

In the scheduling, we will use a partitioning of the processor/time diagram as shown on figure 2. The parts denoted by "First shelf" and "Second shelf" will respectively be associated to the sets of tasks P_1 and P_2. The third part will be associated with the set P_{full}. They will be explain in the next section.

3 Partitioning

The principle of the algorithm is described as follows. The approach is divided in two phases.

1. In the first one, we partition the set of tasks to be scheduled into the two sets P_1 and P_2 as in [14], and we set each task in its *canonical allotment* as defined before. The set P_{full} is empty. At the end of this phase we have determined for each task the shelf on which it should be executed and the number of processors it needs to fulfill the expected time bound of the considered stage. Then, we will reduce the allotment for some tasks to prepare the second

phase. Those reductions are only depending on the previous allotment. If the scheduling is not feasible, some transformations are done to the tasks of the first and second shelves.

2. The second phase is the scheduling of the tasks itself. This scheduling is a simpler form of strip packing, since we have some useful properties on the tasks after the previous transformations. With the new allotments we will show that the schedule is feasible within the time bound of $2(1 - 1/k)$. The global structure of the scheduling is the structure shown in figure 2.

3.1 Partitioning the tasks

The partition is done by solving the following knapsack problem with dynamic programming:

$$W^* = \min_{P_1,P_2} \left[\sum_{i \in P_1} W\left(i, p_i^1\right) + \sum_{i \in P_2} W\left(i, p_i^{1/2}\right) \right]$$

With the following constraint:

$$\sum_{i \in P_1} p_i^1 \leq mk$$

The dynamic programming equation is the following:

$$\bar{W}(i,f) = min \begin{pmatrix} \bar{W}(i-1,f) + W\left(i, p_i^{1/2}\right), \\ \bar{W}(i-1,f-p_i^1) + W\left(i, p_i^1\right) \end{pmatrix}$$

with i a task index and f is the number of idle processors on the first shelf. As this is an integer problem, the complexity is $O(nmk)$ where n is the total number of tasks.

The idea behind this partitioning is related to the examination of the optimal solution. Even in the optimal solution, a given processor cannot execute more than one task during more than a half of the optimal completion time. Therefore there is a natural partitioning of the tasks in two sets: those which need more than half of the optimal computation time and those which do not. The partitioning obtained after solving the knapsack problem above is not necessarily connected with the optimal schedule. As this solution is a minimum (which may not satisfy the capacity constraint of P_2), the total workload W^* is smaller than the workload W^{opt} of the optimal. As we assumed that the optimal makespan is less than 1, we have $W^{opt} \leq mk$ where mk is the total number of processors.

Thus, if $W^* > mk$ there is no feasible schedule with completion time lower than 1 (the guessed optimal makespan was wrong), otherwise we can proceed to the next steps to achieve a guaranteed schedule.

3.2 Reducing the allotment

As mentioned before, scheduling efficiently the tasks on a cluster without breaking the *best placement* rule is generally not easy.

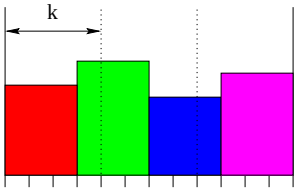

Figure 3: Example with 4 tasks $m = 3$, $k = 4$.

To be able to schedule efficiently with respect to the *minimal penalty* assumption we have to change slightly the allotment of the tasks. According to the monotony assumption, reducing the allotment does not increase the workload. On the opposite increasing the allotment may have unexpected results, so we will only accept transformations which reduce it. In the example of figure 3, reducing the allotment of one task on a single processor is one of the possible solution to achieve a schedule which satisfies the best placement. However this solution is probably one of the worst according to the makespan. A better transformation is presented on the left of figure 4, where all the tasks that were on 3 processors are reallocated on 2 processors only. This solution can be compared to the stacking of the two smallest task shown on the right of the same figure.

The generalization of this example is the following:

For all task T_i allotted to the first set of processors (P_1), let a_i and b_i be integers with $b_i < k$ verifying $p_i^1 = a_i k + b_i$. If $b_i \neq 0$ let j_i be the largest integer verifying $2^{j_i} \leq b_i$. The new allotment is

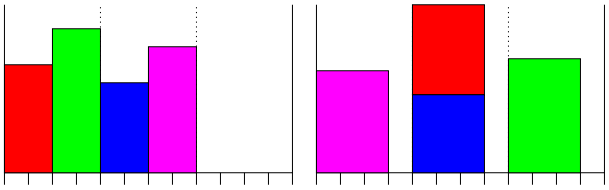

Figure 4: Rearrangements of figure 3.

$Alloc(i) = a_i k + 2^{j_i}$. If $b_i = 0$ the allotment remains unchanged. For instance, with $k = 8$, if $p_i^1 = 15$, the task will be allotted to $8 + 4 = 12$ processors. With $k = 4$ the same task is allotted to $3 \times 4 + 2 = 14$ processors.

All the tasks whose allotment have been reduced are placed in a new set P_{full} corresponding to the area called *full time processors* in figure 2. We can prove the two following useful lemmas on the completion times of the tasks of P_{full}.

Lemma 1 *The reduced tasks are executed in at most* $2(1 - 1/k)$ *units of time.*

Proof. The monotony assumption on the task implies $W(i, r) \leq W(i, s)$ if $r \leq s$. So we have:

$$t_{i,r} r \leq t_{i,s} s$$

$a_i k + b_i$ being the canonical allotment, we also have $t_{i,a_i k + b_i} \leq 1$ for all i. Replacing here r and s by the old and new allotment we can write:

$$t_{i,a_i k + 2^{j_i}} (a_i k + 2^{j_i}) \leq a_i k + b_i$$

And with some straightforward majorations, the result holds:

$$t_{i,a_i k + 2^{j_i}} \leq \frac{a_i k + b_i}{a_i k + 2^{j_i}} \leq \frac{b_i}{2^{j_i}}$$

$$t_{i,a_i k + 2^{j_i}} \leq \frac{2^{j_i + 1} - 1}{2^{j_i}}$$

$$t_{i,a_i k + 2^{j_i}} \leq \frac{2^q - 1}{2^{q-1}} = 2(1 - \frac{1}{k})$$

□

Lemma 2 *All the tasks in* P_{full} *have a completion time larger than* 1 *(unit of time).*

Proof. For all tasks in P_{full}, the allotment is strictly smaller than the canonical allotment, which is the smallest allotment to have a completion time less than or equal to 1. □

4 Principle of the allotment

Let us now detail the principle of the allotment in each area. In order to avoid useless manipulations of the tasks, the actual scheduling is only done at the end of the algorithm, when all the necessary transformations have been done.

4.1 Allotment in P_{full}

The problem is to allocate all the tasks of P_{full} with the minimum of wasted space. To achieve this goal, we consider again the allotment as $Alloc(i) = a_i k + 2^{j_i}$. The allotment of the $a_i k$ part is easy: just by allocating those on the first available SMPs. The remaining 2^{j_i} processors have to be allocated on a single SMP. To do this efficiently, we sort all the remainders of the tasks of P_{full} according to j_i. Then we allocate all the tasks from left to right.

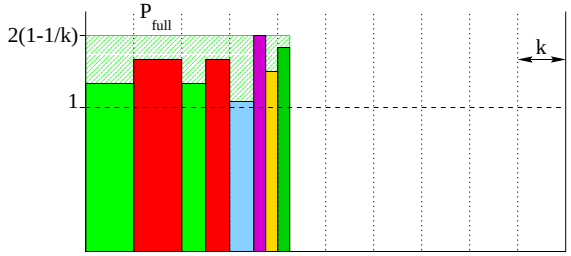

Figure 5: Result of the allotment in P_{full}.

Lemma 3 *This allotment respects the minimal penalty hypothesis.*

Proof.
We need to prove that for all i, $1 \leq i \leq m - 1$, there is no task overlapping the gap between processors of index ki and $ki + 1$. Or in other words, processors ki and $ki + 1$ are not allotted to the same task. This is quite simple as the sorting of the remainders ensures that for a task of size 2^j all the tasks placed on its left have a size which is a multiple of 2^j, since all the sizes are powers of 2.

Considering an overlapping task of size 2^j we can define an integer l such as $l2^j + 1$ is the first processor allotted to the task and $(l + 1)2^j$ is the last.

If the processors ki and $ki + 1$ are allotted to this task, the processor ki is between $l2^j + 1$ and

$(l+1)2^j - 1$. This interval modulo 2^j is $[1; 2^j - 1]$ and ki modulo 2^j is 0. Therefore the gap cannot be during the execution of a task, which concludes the proof. □

4.2 Allotment in P_1

The allotment of P_1 is close to the allotment of P_{full}, the only difference between the tasks of both sets is their completion times. All the tasks have an allotment on $a_i k + 2^{j_i}$ processors, and the same tactic works. The only difference is that here the processors are filled from right to left.

Lemma 4 *The total number of processors needed for P_{full} and P_1 is less or equal to mk*

Proof. The constraint $\sum_{i \in P_1} p_i^1 \leq mk$ on the partitioning ensures that the total number of processors needed for the tasks of the first shelf is less or equal to mk. Since all the tasks in P_{full} are tasks that were initially in P_1, whose allotment has been reduced, the total number of processors needed by the two sets is still less or equal to mk. □

The above lemma ensures that there is no overlap between tasks in P_{full} and P_1. At this stage the allotment looks like as depicted in figure 6.

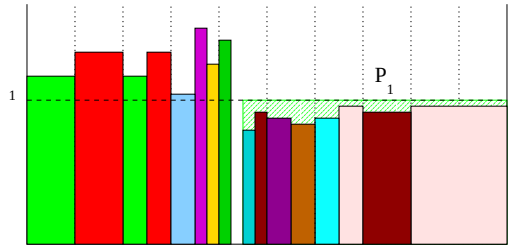

Figure 6: Schematic representation of P_{full} (left) and P_1 (right). Some processors may be idle (white).

4.3 Allotment in P_2

If we want to build a solution within the $2(1 - 1/k)$ bound, we cannot reduce the allotment of the tasks of P_2 as it was done for P_1. However, we will see that we can afford to lose some processors and obtain a good allotment anyway. For each task i in P_2 allocated on $a_i k + b_i$ processors by the canonical allotment, we may change the allotment to $a_i k + 2^{j_i}$, where j_i is the smallest integer verifying $b_i \leq 2^{j_i}$.

Formally for the proofs in the rest of the paper, we will assume that the tasks of size $a_i k + b_i$ are put in boxes of size $a_i k + 2^{j_i}$, without changing their allotment (i.e. $2^{j_i} - b_i$ processors are idle). The proofs are based on the workload, therefore we cannot afford to increase it.

The allotment of the boxes is the same as above, separating the parts using entirely SMPs, and sorting the remainders to allocate them from right to left in the second shelf. An example of this allotment is presented on figure 7.

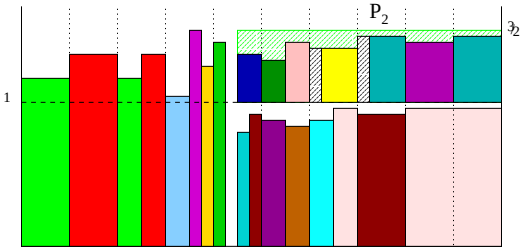

Figure 7: Representation of P_2. Striped areas are idle processors induced by tasks using $\frac{3}{4}k$ processors.

This schedule may be not feasible, *i.e.* there might be more processors needed for the second shelf than available. To manage to schedule all the tasks, we need to make some transformations described in section 5.

Lemma 5 *The workload of a box of size 2^j is at least $2^{j-2} = \frac{2^j}{4}$*

Proof. The allotment of a task T_i put in a box of size 2^j is necessarily greater than or equal to $2^{j-1} + 1$, which means that using 2^{j-1} processors it will be executed in a time $t_{i,2^{j-1}}$ greater than $1/2$. Then the workload $W(i, b_i)$ of this task on b_i processors is verifying:

$$W(i, b_i) \geq 2^{j-1} t_{i,2^{j-1}} > 2^{j-2}$$

In the final analysis we will need a stronger version of this lemma, proving that we can achieve a 75% occupation time for the boxes when there is at least one free processor on the first shelf. When k is greater than or equal to 8, we can have tasks in P_2 running in at most $\frac{3}{4}$ units of time without

overpassing the $2(1 - 1/k)$ time bound. This is why we may reduce the allotment of the tasks which are using at most 3 out of 4 processors in their boxes to the lower box size.

When k is equal to 4 this is not possible. However, as we will see later if there is an idle processor on the first shelf, all the tasks in P_2 are using 4 processors or more. That is, the smallest boxes with unused processors are boxes of size 8 with tasks using only 7 processors, which is enough to fill 75% of the box. □

4.4 Properties of the allotment

The three sets described above, namely P_{full}, P_1 and P_2, are used for allotment, but we can also prove some useful properties. The most important one concerns the workload of each set. Let us denote in the rest of the paper s_1, s_2 and s_{full} the respective number of processors used by the sets of tasks P_1, P_2 and P_{full} ($s_1 + s_{full} \leq mk$). Lemma 2 shows that the workload of P_{full} is at least equal to s_{full}. Similarly the number of tasks in P_2 is noted n_2.

The partitioning ensures that all the processors used in P_1 work during at least $\frac{1}{2}$ unit of time. Therefore the workload of this part is at least $\frac{1}{2}s_1$. The transformations in the following section improves this processor occupation up to $\frac{3}{4}s_1$.

Finally, if we remove all the sequential tasks (i.e. tasks allotted to 1 processor) from P_2 we can establish from property 1 that all the tasks in this set have an execution time greater than or equal to $\frac{1}{4}$, because they are all executed on at least two processors. The placement of the sequential tasks of P_2 is done in the following way:

As we know that the total workload is lower than or equal to mk, we know that there is at least one processor which is not working for more than 1 unit of time. If we have a small sequential task to place, it can be put on top of the tasks of the first shelf, with a potential delay in the starting time of the task of the second shelf scheduled on the processor (if there is one). As the execution time of the small sequential is less or equal to $\frac{1}{2}$, the total workload done on the processor is less or equal to $\frac{3}{2}$. Therefore the task of the second shelf will not end after the $\frac{3}{2}$ time bound. This can be repeated until all the small sequential tasks have been placed.

For all the following proofs based on the workload, the workload of the small sequential tasks (denoted W_{seq}) can be added to $W_{P_{full}}$.

5 Transformations

In this section we detail three transformations to move tasks from one set (P_2 or P_1) to another (P_1 or P_{full}). The common asset of these transformations is that they don't increase the total workload. These transformations can be done in any order and as often as required.

5.1 Moving a task from P_2 to P_1 or P_{full}

Let f be the number of idle processors between P_1 and P_{full} on the first shelf ($f = mk - s_1 - s_{full}$). Let T_i be a task in P_2 for which $p_i^1 = a_i k + b_i$ and $a_i k + 2^{j_i} \leq f$ where j_i is the largest integer such as $2^{j_i} \leq b_i$. T_i is moved from P_2 to either P_1 or P_{full} depending on its completion time, and it still satisfies the properties of lemmas 1, 2 and 4.

If there is only one remaining task in P_2 and the schedule is still not feasible (i.e. $s_{full} + s_2 > mk$), we can remove this task from the second shelf to put it on the first shelf. This last task may need all the idle processors of the first shelf, therefore we have to put them according to the *best placement* rule. As there is no more second shelf ($P_2 = \emptyset$) the set P_1 and P_{full} can be mixed as depicted in figure 8. All

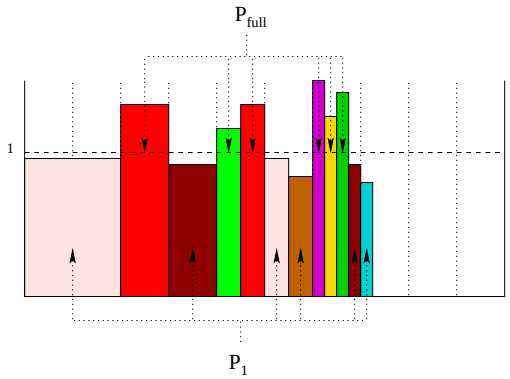

Figure 8: The idle processors are grouped in a good configuration.

the idle processors are then gathered on the right

border of the processor-time diagram, and the last task can be allotted in its *best placement*.

Similarly, let us consider the case where there remains two tasks in P_2 and one has to be placed on the f idle processors. We cannot as previously mix all the tasks of P_1 and P_{full}, because we need to keep as many processors usable for the last task remaining in P_2. Therefore, we will just group the tasks of P_1 and P_{full} which do not fill a SMP as shown in figure 9.

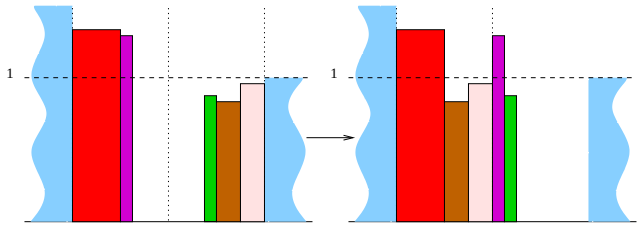

Figure 9: Case where the idle processors are in a "good" configuration.

5.2 Moving a task from P_1 to P_{full}

Reducing a task allotment from $a_ik + 2^{j_i}$ to $a_ik + 2^{j_i-1}$ allows to keep 2^{j_i-1} processors available for other tasks, doubling at most the task completion time. This transformation can then be applied to all the tasks in P_1 with completion time less than or equal to $1 - 1/k$ units of time. This transformation is only applied to tasks with completion time less than $3/4$. This means that only tasks scheduled on two processors and with execution time lower than $3/4$ are concerned, because all the tasks for which $p_i^1 \geq 4$ have an execution time greater than this bound[1].

This transformation cannot of course be applied to sequential tasks, which had a special treatment.

Lemma 6 *All tasks in P_1 allocated to more than 1 processors have a completion time greater than $3/4$ when this transformation is not possible.*

Proof. This transformation cannot be applied if a task T_i is allocated on exactly a_ik processors. The monotony of malleable tasks ensures:

$$W_{(i,a_ik)} \geq W_{(i,a_ik-1)}$$

$$t_{i,a_ik}\ a_ik \geq t_{i,a_ik-1}\ (a_ik - 1)$$

And as $a_ik = p_i^1$, we have $t_{i,a_ik-1} > 1$.

$$t_{i,a_ik} > 1 - \frac{1}{a_ik} \geq 1 - \frac{1}{k} \geq 1 - \frac{1}{4}$$

□

5.3 Stacking two sequential tasks from P_1 to P_{full}

If there are two sequential tasks of completion time less or equal to $1 - 1/k$, we can put one on top of the other in P_{full}. As the small sequential tasks with completion time less than $1/2$ units of time has been removed (section 4.4), this stacking will have a completion time greater than 1 and lower than $2(1 - 1/k)$.

Claim 1 *The total workload of P_1 is strictly greater than $\frac{3}{4}(s_1 - 1) + \frac{1}{2}$.*

When this transformation is no more possible, there is at most one (sequential) task in P_1 with a completion time less than $3/4$. As we ruled out the small sequential tasks, this task takes more than $1/2$ units of time. □

6 Analysis

At this point the schedule is almost always feasible. However, some pathological cases may need one final extra move.

Let us assume that the schedule is still not feasible when none of the previous transformations can be applied. As the following proofs use an argument on the workloads of the different parts of the schedule, we have to recall some properties on these workload.

Recall that $f = mk - s_1 - s_{full}$ is the number of idle processors on the first shelf. The total workload is the sum of the workload in the three parts:

$$W_{total} = W_{P_1} + W_{P_2} + W_{P_{full}}$$

Property 3 $W_{total} < mk$ comes from section 3.1 .

[1] Except for the special case k=2 where the tasks scheduled on 3 processors might also be reduced by this transformation. But this case will be considered apart

Property 4 $W_{P_{full}} > s_{full}$ comes from lemma 2 .

Let x be a boolean variable equal to 1 if there is in P_1 a sequential task with execution time lower or equal to $\frac{3}{4}$ units of time, and 0 if there is not.

Property 5 $W_{P_1} > \frac{3}{4}s_1 - \frac{x}{4}$.

It comes from section 5.3, and from lemma 6 if there is no sequential smaller than $\frac{3}{4}$ units of time. Combining property 3 with properties 4 and 5 we get the following expression on W_{P_2}:

$$f + \frac{s_1 + x}{4} > W_{P_2} \quad (1)$$

However, as we know that the schedule is still not feasible, and that each task of the set P_2 has an execution time greater than $\frac{1}{4}$, we can also write:

$$W_{P_2} > \frac{f + s_1 + 1}{4}$$

Therefore:

$$\begin{aligned} f + \frac{s_1 + x}{4} &> \frac{f + s_1 + 1}{4} \\ \frac{3}{4}f &> \frac{1 - x}{4} \geq 0 \end{aligned}$$

And as f is an integer, f is at least equal to 1.

We have more informations on the tasks in P_2 since the schedule is not feasible. We can also write two different lower bounds for W_{P_2}:

Property 6 $W_{P_2} > \frac{3(f+s_1+1)}{8}$

Proof. The proof of this property relies on the processor occupation time for P_2. In each box, the workload is greater than 75% of the box area, as said in section 4.3. If the schedule is not feasible, the boxes used at least $s_1 + f + 1$ processors, and therefore we can write the previous inequality. □

Property 7 $W_{P_2} > n_2 \frac{f+1}{2} \frac{3}{2}$

Proof. This property just states that all the n_2 tasks in P_2 could not be allocated in P_{full} by the second transformation, which means that they use at least $\frac{f+1}{2}$ processors for more than $\frac{3}{2}$ units of time. □

Moreover, if there is only one task in P_2 it could not be placed on f processors, which means that its workload is greater than $3/2 f$. This fact combined with the above property allows to write that $W_{P_2} > 3/2 f$ in all cases.

Along with equation 1, this inequality implies that $s_1 + x > 2f$.

From these properties 6 and 7, both combined with equation 1, we have the two following inequalities:

$$\begin{aligned} 5f + 2x - 3 &> s_1 \quad (2) \\ s_1 + x &> 3n_2(f + 1) - 4f \quad (3) \end{aligned}$$

Combining these both we obtain $n_2 < 3$, which leads to distinguish only two cases. These cases are technical, and the proofs are presented in the appendix. In both cases one ultimate transformation makes the schedule feasible.

7 Scheduling

Once the allotment is determined, the scheduling is the following:

- Tasks of P_1 and P_{full} are starting at $t = 0$,
- Inserted sequential tasks then start as soon as possible,
- Tasks of P_2 start as soon as possible.

8 Algorithm

The algorithm can be summarized in the following steps:

1. The partition of tasks in P_1 and P_2.
2. If the total workload exceeds mk, then exit with an error (the guessed optimal is wrong).
3. Remove the sequential tasks from P_2.
4. Reduce the tasks in P_1 which are not in an $a_i k + 2^{j_i}$ allotment and put them in P_{full}.
5. While the scheduling is not feasible and some transformations can be made, transform the allotment.
6. If the scheduling is still impossible we are in one of last cases: use the final transformation.
7. Insert the small sequential tasks.
8. Schedule the tasks and output successfully.

Complexity

The partitioning is done in $O(nmk)$. The extraction of the small sequential tasks is done in $O(n)$. The first reduction (before the loop) is done in $O(n)$ at worst since we just have to consider the tasks in P_1 and look if their allocation is correct. The loop is also executed at $O(n)$ times, since each task can undergo at most two transformations. Finding a task which can be used for one of the transformation is feasible in time $O(m \log_2 k)$.

Sorting the tasks in the three sets is done in $O(n)$ and so is the scheduling. Finally the insertion of the sequential is at worst in time $O(nmk)$.

Therefore the complexity of the whole algorithm is $O(nmk)$.

9 Special case $k = 2$, clusters of biprocessors

The bound $2(1 - 1/k)$ can obviously not be reached for $k = 2$. However we can reach the bound of $3/2$ which is the best currently known with polynomial approximation scheme for independent malleable tasks [13]. The main idea is to take the solution given by the algorithm in the non hierarchical case, and to place correctly all the tasks to comply with the best placement rule. With $k = 2$ the only placement problem is when tasks allocated on an even number of processors are misaligned with SMPs. A simple way to deal with this problem is to place all the even sized task on the extremities of the processor-time diagram. The three parts P_1, P_2 and P_{full} being defined as before, we just have to place the tasks running on an even number of processors on the left for P_{full} and on the right for P_1 and P_2 as depicted in figure 10.

The only difficulty is with the final transformation used in the non hierarchical problem. This transformation is the "extra move" we presented in the analysis in the case $n_2 = 1$. In the non hierarchical case, stacking the sequential on the smallest task of P_1 is not a problem because there is no *minimal penalty* constraint. However here we cannot insert a task in the middle of P_2 and still guarantee the *best placement* for all tasks. Hopefully, it is possible to prove as we did for $k > 2$ that this transformation is only applied when there is just one task in P_2.

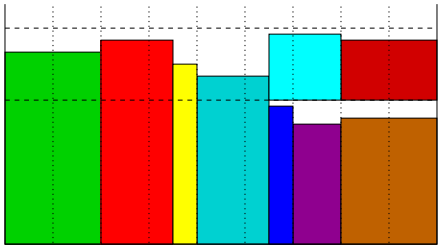

Figure 10: Tasks are on contiguous processors.

10 Concluding remarks and perspectives

In this article, we were looking for fast algorithms that are good approximation for solving the problem of scheduling independent static MT on an homogeneous hierarchical system. This reflection was based on the analysis of the best algorithm known for the non hierarchical problem. We show for the problem of scheduling independent static MT that the same approach is very useful for hierarchical systems. The algorithm presented here reach the same bound as for the non hierarchical problems in the cases $k = 2$ and $k = 4$.

Among problems which remain to be solved for designing efficient software tools for managing efficiently the parallelism of applications in clusters, let us emphasize the problem of heterogeneity which is more complicated under the MT model since it requires non monotonic assumptions.

Finally, on-line policies are needed in many cases where the applications are not fully predictable or in a multi-users environment where some pieces of work on static MT can be used as starting points.

References

[1] R. Baker, E.G. Coffman, and R.L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[2] E. Blayo, L. Debreu, G. Mounie, and D. Trystram. Dynamic load balancing for ocean circulation model with adaptive meshing. In *EuroPar'99*, volume 1685 of *LNCS*. Springer-Verlag, 1999.

[3] R. Blumafe and D.S. Park. Scheduling on networks of workstations. *3rd Intel. Symp. of High Performance Distr. Computing*, pages 96–105, 1994.

[4] E.G. Coffman, M.R. Garey, D.S. Johnson, and R.E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[5] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture - A Hardware,Sofware approach*. Morgan Kaufmann Pub., 1999.

[6] T. Decker and W. Krandick. Parallel real root isolation using the Descartes method. In Banerjee et al., editor, *Proceedings of HiPC99*, volume 1745 of *LNCS*, pages 261–268. Springer-Verlag, 1999.

[7] D. Feitelson and L. Rudolph et al. Theory and practice in parallel job schedulin. In *Workshop on Job Scheduling Strategies for Parallel Processing, IPPS'97*, volume 1291 of *LNCS*, pages 1–34. Springer-Verlag, 1997.

[8] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman, New York, 1979.

[9] R. Giroudeau. *L'impact des délais de communications hiérarchiques sur la complexité et l'approximation des problèmes d'ordonnancement*. PhD thesis, Université d'Évry, December 2000. in french.

[10] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[11] K. Jansen and L. Porkolab. Linear time approximation schemes for scheduling problems. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 99*, pages 490–498, 1999.

[12] W. T. Ludwig. *Algorithms for scheduling malleable and nonmalleable parallel tasks*. PhD thesis, University of Wisconsin - Madison, Department of Computer Sciences, 1995.

[13] G. Mounié. *Efficient scheduling of parallel application : the monotonic malleable tasks*. PhD thesis, Institut National Polytechnique de Grenoble, June 2000. in french.

[14] G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithm for scheduling malleable tasks. In *Proc. of 11th ACM Symposium of Parallel Algorithms and Architecture*, pages 23–32, 1999.

[15] G. Mounie, C. Rapine, and D. Trystram. A $\frac{3}{2}$-approximation algorithm for independent scheduling malleable tasks. submitted for publication 2001.

[16] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[17] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.

[18] T. Sterling and D. Becker et al. Beowulf: A parallel workstation for scientific computation. In *ICPP'95*, volume 1, pages 11–14, 1995.

[19] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, 1992.

Appendix

The two cases from section 6 are described in detail here:

Case $n_2 = 1$

If there is only one task in P_2, and the schedule is still impossible, we know that this task T_i needs more than all the free processors in P_2.

$$W_{P_2} = t_i p_i^{1/2} > \frac{p_i^{1/2} - 1}{p_i^{1/2}} p_i^{1/2} = p_i^{1/2} - 1 \geq \frac{f + s_1}{2}$$

And then as we have $f + \frac{s_1 + x}{4} > W_{P_2}$ we have:

$$\begin{aligned} f + \frac{s_1 + x}{4} &> \frac{f + s_1}{2} \\ 2f + x &> s_1 \end{aligned}$$

As we already proved $s_1 + x > 2f$, we necessarily have $x = 1$ and $s_1 = 2f$. The extra move needed here is to remove the sequential task running in time less than $\frac{3}{4}$ units of time from P_1, stack it over the smallest task of P_1, and place the last task of P_2 on the $f + 1$ free processors.

To guarantee the $2(1 - 1/k)$ bound, we have to prove first that W_{P_1} is lower than $3/4 s_1$.

Assume that $W_{P_1} \geq 3/4 s_1$. The equation 1 can now be written $f + \frac{s_1}{4} \geq W_{P_2}$. With $s_1 = 2f$ we have $3/2 f \geq W_{P_2}$ which contradicts $W_{P_2} > 3/2 f$

Now let r_1 and r_2 be strictly positive real number with $1/2 + r_1$ being the completion time of the sequential and $3/4 + r_2$ being the completion time of the smallest task of P_1.

$$\begin{aligned} W_{P_1} &\geq 1/2 + r_1 + (s_1 - 1)(3/4 + r_2) \\ 3/4 s_1 &> W_{P_1} \\ 1/4 &> r_1 + (s_1 - 1) r_2 \end{aligned}$$

The stacking of the sequential task and the smallest one of P_1 will have a completion time lower than $3/2$.

The final proof we need is that there exists another task in P_1 than the removed sequential one. If the sequential task is the only one in P_1, we can write from equation 1:

$$f + \frac{1}{2} > W_{P_2} > \frac{3}{2} f$$

By contradiction, as $f \neq 0$ there is another task in P_1.

Last but not least, the last task of P_2 fits in the first shelf with $f+1$ processors, because $3/2(f+1) > f + \frac{s_1+1}{4} > W_{P_2}$.

Case $n_2 = 2$

As we didn't prove that this case cannot happen, we provided another tricky transformation to deal with it. First let us prove that the biggest task of the two can be placed on the f free processors and it is still completed in time less than $\frac{3}{2}$ units of time.

If the biggest task cannot be placed on f processors, its workload is greater than $\frac{3}{2} f$. As the workload of the small task is greater than $\frac{3}{4}(f + 1)$ we have:

$$\begin{aligned} W_{P_2} &> \frac{3}{2} f + \frac{3}{4}(f + 1) = \frac{9f + 3}{4} \\ f + \frac{s_1 + x}{4} &\geq W_{P_2} \\ 4f + s_1 + x &> 9f + 3 \\ s_1 + x &> 5f + 3 \end{aligned}$$

Which contradicts equation 2. So the biggest task can be placed on the f free processors. Now we will prove that there is a way to schedule the tasks of P_1 and P_{full} and still have enough place to put the last task of P_2 in its best placement.

If the task scheduled on f processor lasts after one unit of time, its workload is greater than f, and from equation 1 we deduce that the smallest has a workload lower than $\frac{s_1}{4}$, and then can be scheduled on the second shelf on at most $\frac{s_1}{2}$ processors.

If the task scheduled on f processor does not lasts for more than one unit of time, those f processors can be used to schedule the last task of P_2. First let us prove that the smallest task has a workload lower than or equal to $\frac{f+s_1}{4}$. If it did not, we would have $W_{P_2} > \frac{f+s_1}{2}$ and with equation 1 we would have $2f + x > s_1$ which as we have proved before implies $x = 1$ and $s_1 = 2f$. But in the case $n_2 = 2$ we also have the equation 3 which says that $s_1 > 2f + 6 - x$ and leads to a contradiction.

Then the smallest task has a workload lower than or equal to $\frac{f+s_1}{4}$, and can be scheduled on at most $\frac{s_1+f}{2}$ processors on the second shelf.

In both cases the allotment shown in figure 9 ensures that there is enough free processors in the good configuration. If there are some SMPs completely idle over P_1 or over the last task inserted, they have a total number of processors greater than (respectively) $\frac{s_1}{2}$ or $\frac{s_1+f}{2}$. If there are no idle SMP on the second shelf, the last task of P_2 can be scheduled on one of the two remaining SMP shown in figure 9.